Declarative Programming of Interactive Systems with Grapefruit

Wolfgang Jeltsch

Brandenburgische Technische Universität Cottbus
Cottbus, Germany

Software Technology Colloquium at Utrecht Universiteit
May 29, 2008

Grapefruit overview

- Haskell library for Functional Reactive Programming with a focus on GUIs and animated graphics
- push-based implementation for efficiency
- based on Gtk2Hs and HOpenGL, multi-toolkit support planned
- currently developed at the BTU Cottbus
- several Cabal packages:
  - *grapefruit-frp* Functional Reactive Programming core
  - *grapefruit-records* record system
  - *grapefruit-gui* GUI programming
  - *grapefruit-graphics* graphics programming
  - *grapefruit-examples* illustrations of Grapefruit's features
- homepage at `http://haskell.org/haskellwiki/Grapefruit`

This talk

- outline:
  1. Functional Reactive Programming
  2. Record system
  3. Multi-toolkit support
- everything presented here either implemented or to be implemented in the foreseeable future (hopefully)

Signals

- describe behavior over time
- several kinds of signals:

discrete	values at discrete times
continuous	values at all times
segmented	values at all times, discrete times where changes may occur

- examples of signals:

discrete	sequence of key presses (type *DSignal Char*)
continuous	time (type *CSignal DiffTime*)
segmented	title of a window (type *SSignal String*)

- class *Signal* with instances for *DSignal*, *CSignal* and *SSignal*

Circuits

- circuits for communicating with the real world
- have inputs and outputs which are tuples of signals
- kinds of circuits:
    - plain circuits

$$\mathit{timeProvider} :: \mathit{PlainCircuit}\ ()\ (\mathit{CSignal}\ \mathit{DiffTime})$$

    - circuits with additional “effects”
        - widget circuits
        - window circuits
- circuit composition using arrow methods

Arrows

- arrows informally:
  - effectful computations like monads
  - explicit input (contrary to monads)
  - more general than monads
- arrow syntax (simplified):
  - **proc**-expressions

$$
\begin{array}{l}
\textbf{proc}\ \langle \mathit{input} \rangle \rightarrow \textbf{do} \\
\quad \langle \mathit{output}_1 \rangle \leftarrow \langle \mathit{arrow}_1 \rangle \prec \langle \mathit{input}_1 \rangle \\
\quad \vdots \\
\quad \langle \mathit{output}_n \rangle \leftarrow \langle \mathit{arrow}_n \rangle \prec \langle \mathit{input}_n \rangle \\
\quad \mathit{returnA} \prec \langle \mathit{output} \rangle
\end{array}
$$

  - **rec**-blocks for constructing arrows with cycles

Switching and signal instantiation

- family of functions for switching between different signals

$$
\begin{aligned}
\mathit{switch} ::\ & (\mathit{Signal}\ \mathit{signal}) \\
& \Rightarrow \mathit{SSignal}\ (\mathit{signal}\ \mathit{val}) \rightarrow \mathit{signal}\ \mathit{val}
\end{aligned}
$$

- different signal consumers run their own signal instance
- signal is instantiated when
  - a circuit is created which consumes the signal
  - the signal is switched into
- behavior may depend on instantiation time if signal is stateful
- memoization and early starting:

$$
\begin{aligned}
\mathit{withSignal} ::\ & (\mathit{Signal}\ \mathit{signal}, \mathit{Signal}\ \mathit{signal}') \\
& \Rightarrow \mathit{signal}\ \mathit{val} \rightarrow (\mathit{signal}\ \mathit{val} \rightarrow \mathit{signal}'\ \mathit{val}') \\
& \rightarrow \mathit{signal}'\ \mathit{val}'
\end{aligned}
$$

Functions on discrete signals

- some discrete signal functions:

$$
\begin{aligned}
\mathit{fmap} &:: (\mathit{val} \rightarrow \mathit{val}') \rightarrow (\mathit{DSignal}\ \mathit{val} \rightarrow \mathit{DSignal}\ \mathit{val}') \\
\mathit{filter} &:: (\mathit{val} \rightarrow \mathit{Bool}) \rightarrow (\mathit{DSignal}\ \mathit{val} \rightarrow \mathit{DSignal}\ \mathit{val}) \\
\mathit{scan} &:: (\mathit{val}' \rightarrow \mathit{val} \rightarrow \mathit{val}') \rightarrow \mathit{val}' \\
&\rightarrow (\mathit{DSignal}\ \mathit{val} \rightarrow \mathit{DSignal}\ \mathit{val}') \\
\mathit{merge} &:: \mathit{DSignal}\ \mathit{val}_1 \rightarrow \mathit{DSignal}\ \mathit{val}_2 \\
&\rightarrow \mathit{DSignal}\ (\mathit{MergeVal}\ \mathit{val}_1\ \mathit{val}_2)
\end{aligned}
$$

$$
\begin{aligned}
\mathbf{data}\ \mathit{MergeVal}\ \mathit{val}_1\ \mathit{val}_2 &= \mathit{First}\ \mathit{val}_1 \\
&\mid \mathit{Second}\ \mathit{val}_2 \\
&\mid \mathit{Both}\ \mathit{val}_1\ \mathit{val}_2
\end{aligned}
$$

Functions on segmented signals

- some segmented signal functions:

$$
\begin{aligned}
\mathit{initAndHold} &:: \mathit{val} \to \mathit{DSignal}\ \mathit{val} \to \mathit{SSignal}\ \mathit{val} \\
\mathit{fmap} &:: (\mathit{val} \to \mathit{val}') \\
&\to (\mathit{SSignal}\ \mathit{val} \to \mathit{SSignal}\ \mathit{val}') \\
\mathit{pure}, \mathit{return} &:: \mathit{val} \to \mathit{SSignal}\ \mathit{val} \\
\mathit{join} &:: \mathit{SSignal}\ (\mathit{SSignal}\ \mathit{val}) \to \mathit{SSignal}\ \mathit{val} \\
(\langle * \rangle) &:: \mathit{SSignal}\ (\mathit{val} \to \mathit{val}') \\
&\to (\mathit{SSignal}\ \mathit{val} \to \mathit{SSignal}\ \mathit{val}') \\
(|*|) &:: \mathit{SSignal}\ (\mathit{val} \to \mathit{val}') \\
&\to (\mathit{SSignal}\ \mathit{val} \to \mathit{SSignal}\ \mathit{val}')
\end{aligned}
$$

Functions on continuous signals

- some continuous signal functions:

$$
\begin{aligned}
\mathit{fmap} &:: (\mathit{val} \rightarrow \mathit{val}') \\
&\rightarrow (\mathit{CSignal}\ \mathit{val} \rightarrow \mathit{CSignal}\ \mathit{val}') \\
\mathit{pure} &:: \mathit{val} \rightarrow \mathit{CSignal}\ \mathit{val} \\
(\langle * \rangle) &:: \mathit{CSignal}\ (\mathit{val} \rightarrow \mathit{val}') \\
&\rightarrow (\mathit{CSignal}\ \mathit{val} \rightarrow \mathit{CSignal}\ \mathit{val}') \\
\mathit{withoutSegs} &:: \mathit{SSignal}\ \mathit{val} \rightarrow \mathit{CSignal}\ \mathit{val} \\
\mathit{withSegs} &:: \mathit{DSignal}\ () \rightarrow \mathit{CSignal}\ \mathit{val} \rightarrow \mathit{SSignal}\ \mathit{val}
\end{aligned}
$$

Representing continuous sources

- If there is a way to be notified about changes (events, interrupts) than use segmented signals.
    - no polling
    - no delayed reaction
- If there is no notification mechanism than use continuous signals (even if the signal changes only at discrete points in time).
    - sampling gives the times when to read
    - user can choose among a variety of sampling schemes (clocked, sample when idle, etc.)

Records

- attribute names as first-class values

$$\textbf{data}\ \mathit{Caption} = \mathit{Caption}$$
$$\textbf{data}\ \mathit{IsEnabled} = \mathit{IsEnabled}$$

- records as heterogenous lists of name-value pairs whose structure is mirrored by their types
  - () for empty records
  - strict pair type using infix notation for non-empty records

$$\textbf{data}\ \mathit{init} :\&\ \mathit{last} = !\mathit{init} :\&\ !\mathit{last}$$

  - partially strict pair type for name-value pairs

$$\textbf{data}\ \mathit{name} ::: \mathit{val} = !\mathit{name} := \mathit{val}$$

Records example

- example of a record:

$$\begin{aligned} () &:\& \mathit{Caption} &&:= \mathit{pure}\ \texttt{"Ok"} \\ &:\& \mathit{IsEnabled} &&:= \mathit{fmap}\ \mathit{isValid}\ \mathit{input} \end{aligned}$$

- type of this record:

$$\begin{aligned} () &:\& \mathit{Caption} &&::: \mathit{SSignal}\ \mathit{String} \\ &:\& \mathit{IsEnabled} &&::: \mathit{SSignal}\ \mathit{Bool} \end{aligned}$$

Advanced record features

- attribute order independence
- defaulting for input record attributes
- dropping attributes of output records

Realization of advanced record features

- in input records, name now contains optionality information

$$\begin{aligned} () &:\& \ \mathit{Mandatory}\ \mathit{Caption} \ ::: \mathit{SSignal}\ \mathit{String} \\ &:\& \ \mathit{Optional}\ \mathit{IsEnabled} \ ::: \mathit{SSignal}\ \mathit{Bool} \end{aligned}$$

- conversions between internal and external records (basic idea):

class *Input internalInput externalInput* **where**
  inputConv :: *externalInput* → *internalInput*
class *Output internalOutput externalOutput* **where**
  outputConv :: *internalOutput* → *externalOutput*

- circuits offered by the library already contain these conversions
- input expressions and output patterns determine what instance to choose

Advanced record features example

- input type:

$$\begin{aligned}() &:\& \mathit{Mandatory}\ \mathit{Caption} ::: \mathit{SSignal}\ \mathit{String} \\ &:\& \mathit{Optional}\ \ \mathit{IsEnabled} ::: \mathit{SSignal}\ \mathit{Bool}\end{aligned}$$

- output type:

$$\begin{aligned}() &:\& \mathit{Push} ::: \mathit{DSignal}\ () \\ &:\& \mathit{MaybeMousePos} ::: \mathit{SSignal}\ (\mathit{Maybe}\ \mathit{Pos})\end{aligned}$$

- arrow statement:

$$() :\& \mathit{Push} := \mathit{ok} \leftarrow \mathit{button} \dashv () :\& \mathit{Caption} := \mathit{pure}\ \texttt{"Ok"}$$

Representing toolkits

- constructorless types for denoting toolkits

data *GTK*

data *Qt*

data *NCurses*

Declaring interfaces

- classes for parts of the GUI interface

class *Toolkit toolkit* **where**
  type *NativeWidget* :: ∗
  type *NativeWindow* :: ∗
  ...
class (*Toolkit toolkit*) ⇒
    SupportsCommonWidgets toolkit **where**
  button :: *Widget toolkit* ......
  label :: *Widget toolkit* ......
class (*Toolkit toolkit*) ⇒
    SupportsTreeView toolkit **where**
  treeView :: *Widget toolkit* ......

Implementing all interfaces

- complete interface supported:

instance *Toolkit* *GTK* **where** . . .
instance *SupportsCommonWidgets* *GTK* **where** . . .
instance *SupportsTreeView* *GTK* **where** . . .

Implementing some interfaces

- some features not supported by underlying library:

  instance *Toolkit* *NCurses* **where** ...
  instance *SupportsCommonWidgets* *NCurses* **where** ...
  -- no support for tree views in ncurses

- some features not supported because of porter's lazyness:

  instance *Toolkit* *Qt* **where** ...
  instance *SupportsCommonWidgets* *Qt* **where** ...
  -- tree views not yet ported

Advantages over using linking tricks etc.

- usage of multiple toolkits in same application/GHCi session
- no extra tools or compiler options needed for toolkit selection
- dependencies of feature sets can be declared (via subclassing)
- required compliance level encoded in the types
  - compliance level can be checked
  - required compliance level can be inferred

Declarative Programming of Interactive Systems with Grapefruit

Wolfgang Jeltsch

Brandenburgische Technische Universität Cottbus
Cottbus, Germany

Software Technology Colloquium at Utrecht Universiteit
May 29, 2008

Arrow composition